Ultra-Low Power Data Storage for Sensor Networks

GAURAV MATHUR
Google, Inc.
PETER DESNOYERS
Northeastern University
and
PAUL CHUKIU, DEEPAK GANESAN and PRASHANT SHENOY
University of Massachusetts

Local storage is required in many sensor network applications, both for archival of detailed event information, as well as to overcome sensor platform memory constraints. Recent gains in energy efficiency of new-generation NAND flash storage have strengthened the case for in-network storage by data-centric sensor network applications. We argue that current storage solutions offering a simple file system abstraction are inadequate for sensor applications to exploit storage. Instead, we propose Capsule—a rich, flexible and portable object storage abstraction that offers stream, file, array, queue and index storage objects for data storage and retrieval. Further, Capsule supports checkpointing and rollback of object state for fault tolerance. Our experiments demonstrate that Capsule provides platform independence, greater functionality and greater energy efficiency than existing storage solutions.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory technologies*; B.8.2 [**Performance and Reliability**]: Performance Analysis; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Design, Measurement, Performance, Experimentation

Additional Key Words and Phrases: storage system, flash memory, energy efficiency, objects, file system, sensors, embedded systems

This research is supported in part by NSF grants EEC-0313747, CNS-0626873, CNS-0546177, CNS-052072, CNS-0325868, and EIA-0080119.
Preliminary versions of this work appear in Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'06) and the IEEE/ACM Conference on Information Processing in Sensor Networks (IPSN/SPOTS).
Authors' addresses: G. Mathur; email: gmathur@google.com; P. Desnoyers; email: pjd@ccs.neu.edu; D. Ganesan; email: dganesan@cs.umass.edu; P. Shenoy; email: shenoy@cs.umass.edu.

DOI 10.1145/1614379.1614385 http://doi.acm.org/10.1145/1614379.1614385

ACM Reference Format:
Mathur, G., Desnoyers, P., Chukiu, P., Ganesan, D., and Shenoy, P. 2009. Ultra-low power data storage for sensor networks. ACM Trans. Sens. Netw., 5, 4, Article 33 (November 2009), 34 pages. DOI = 10.1145/1614379.1614385 http://doi.acm.org/10.1145/1614379.1614385.

1. INTRODUCTION

Storage is an essential ingredient of any data-centric sensor network application. Common uses of storage in sensor applications include archival storage [Li et al. 2006], temporary data storage [Hellerstein et al. 2003], storage of sensor calibration tables [Madden et al. 2005], in-network indexing [Ratnasamy et al. 2002], in-network querying [Ratnasamy et al. 2001] and code storage for network reprogramming [Hui and Culler 2004], among others. A primary consideration in these applications is the problem of energy efficiency, which has been addressed by a significant fraction of the research in this area. Due to the high relative energy cost of network communication in wireless sensor networks—transmitting a single bit may require as much energy as hundreds of instructions [Hill et al. 2000]—this work has often focused on in-network aggregation and data fusion to reduce radio traffic. This *computation vs. communication trade-off* [Pottie and Kaiser 2000] has had a tremendous influence on the design of both algorithms and platforms for sensor networks. However, the emergence of a new generation of NAND flash storage has added a new variable to this design equation, as sufficient amounts of storage may be used to reduce network traffic by updating the application lazily, deferring transmission of a datum until we are confident that it will be needed by the application. At least one recent study has shown that certain forms of flash storage have combined (write+read) per-byte energy costs *two orders of magnitude less* than total per-byte energy costs (transmit+receive) for low-power radio transmission [Mathur et al. 2006b]. This development challenges existing sensor network design principles, motivating a new set of principles based on trading off not only local computation but local storage as well in order to reduce radio usage and thereby optimize system performance and lifetime.

The emergence of low-cost high-capacity flash storage prompts us to ask: *How can a sensor network storage system be designed to minimize total energy costs for varied sensor applications?* To do this we require a storage system which can provide services tailored to the application, so as to minimize redundant or unnecessary work performed by the application or storage system. This in turn requires a flexible and tunable storage system, which may be adapted to the needs of varied applications for archiving, indexing, and querying.

1.1 Limitations of Existing Flash Storage Systems

Our survey of existing storage solutions (Table I) shows a mismatch between the features they offer and the requirements of sensor applications. We describe in more detail below the limitations we find in these existing sensor storage solutions; to overcome the limitations of these solutions, we develop a new object-based storage system, Capsule, offering rich functionality and flexibility without compromising energy efficiency.

Table I. Comparison of Capsule to Related Efforts

	Storage Devices	Energy Optimized	Memory Optimized	Wear Leveling	Check-pointing	Abstraction	Usage Models
Matchbox	NOR	No	Yes	No	No	Filesystem	File storage; Calibration Tables
MicroHash	MMC	Yes	No	Yes	No	Stream/Index	Stream Storage and Indexing
ELF	NOR	No	Yes	Yes	No	Filesystem	Same as Matchbox
YAFFS	NAND	No	No	Yes	No	Filesystem	Portable devices
Capsule	NAND, NOR	Yes	Yes	Yes	Yes	Object	Data Storage and Indexing; Packet Queues; Temporary Arrays

Mismatch between Storage Abstraction and Application Needs. Many flash-based storage systems, such as YAFFS, YAFFS2 [Manning 2002], Matchbox [Gay 2003] and ELF [Dai et al. 2004], provide a byte-structured file system abstraction to the application. While the traditional rewritable file has come to dominate traditional computing, it often lacks features needed by some sensor applications, or incurs extra operations to implement properties not needed by others. For instance, a common use of local storage is to store a time series of sensor observations and maintain a index on these readings to support queries. If implemented over a byte-structured rewritable file, any overhead needed to provide rewrite capabilities would be wasted, while each application would need to to independently implement both log and index structures. An integrated indexed data stream abstraction, however, would allow multiple applications to take advantage of a single implementation, while that implementation could be carefully tailored to the system storage characteristics.

At least one existing system, MicroHash [Zeinalipour-Yazti et al. 2005], provides such an abstraction. Yet just as a file abstraction is poorly suited to some sensor applications, an indexed stream is poorly suited to others. Examples include "live" application data, where storage is being used as an extension of device memory, calibration tables, and configuration data, to name a few. In these cases, yet other storage abstractions may be best suited to the application's requirements. Rather than advocate a single storage abstraction for all of these cases, we argue that the storage substrate should support a "rich" object storage abstraction with the ability to create, store and retrieve data objects of various types such as files, streams, lists, arrays, and queues, enabling effective use of flash storage by the widest range of applications.

Lack of Portability. Most existing storage solutions have been designed to operate only on a specific type of flash memory—for example, both Matchbox and ELF require capabilities found only in NOR flash but not in NAND. Further, sensor applications written for the same flash memory type do not seamlessly work across different hardware platforms; for example, Matchbox relies on features of a specific flash device, which are not available in most other devices. Capsule, on the other hand, supports portability across both NOR and NAND flash memories, as its design was based on the subset of features common to both these types of flash memories. It organizes the flash as a log and employs a flash abstraction layer to hide the details of the specific flash being used, allowing the higher Capsule layers to be used without any modification on virtually any flash memory.

Lack of Support for Use as a Backing Store. Current flash-based storage systems use flash as a persistent data storage medium. However, memory is often a scarce commodity on small sensor platforms, with Telos and Mica motes containing 10KB and 4KB of RAM, respectively. With empirical studies showing the energy cost of accessing data in flash becoming nearly as low as for data in RAM, it is now possible for applications to use higher-capacity flash for storing live application data and manipulating it in an energy efficient manner. For instance, tasks can use flash as a form of backing store to store large data structures, as well as intermediate results for data processing tasks, enabling the implementation of local data processing algorithms that manipulate data sets larger than the size of RAM.

Files, however, typically do not provide the appropriate storage abstraction for this purpose. It may be nearly impossible to map the data structures of some applications onto append-only streams as provided by, for example, Matchbox, while mapping to rewritable byte-structured files may still be difficult and inefficient. Instead we argue that a richer storage abstraction will allow computational structures to map more easily onto storage, resulting in decreased complexity and increases in efficiency.

Incompatibility with Energy and Memory Constraints. Energy efficiency and the small amount of available memory are key constraints of tetherless sensor platforms—consequently, the storage subsystem for sensor platforms must optimize both constraints. In contrast, traditional nonsensor storage systems have been optimized for bandwidth and access latency, with little regard for memory usage or energy. NAND-flash based file systems such as YAFFS [Manning 2002] are difficult to use on sensor platforms, due to their large RAM footprint. And even in explicitly energy-aware nonsensor file systems, such as BlueFS [Nightingale and Flinn 2004], the target energy usage is far higher than can be sustained by long-lived sensor platforms.

Among existing approaches designed specifically for sensor devices, only MicroHash [Zeinalipour-Yazti et al. 2005] makes claims about energy efficiency. MicroHash, however, requires more memory than is feasible on the sensor platforms which we target, as well as being restricted to a subset of the Capsule functionality.

1.2 Case for In-Network Storage and Archival

Semiconductor-based flash memory is now widely used in applications ranging from BIOS code on motherboards to image storage in digital cameras, and volume production is driving costs down while capacities rise. The emergence of new generations of flash memories has dramatically altered the capacities and energy efficiency of local flash storage. It is possible today to equip sensor devices with several gigabytes of low-power flash storage, while available capacities continue to grow even further.

Further, Table II presents a summary of the results of our detailed measurement study of flash memories [Mathur et al. 2006b]. We notice that new

Table II. Total (device + platform) Amortized Per-byte Energy Consumption of Selected Flash Memories. Data Approximates the Average Energy Needed by the System to Store a Single Byte of Data to Flash and Retrieve it Once

	Per-byte storage cost (μJ)	Platform	Operations measured
Atmel NOR – 0.5MB	6.92	Mica2	write+read
Hitachi MMC – 128MB	1.108	Mica2	erase+write+read
Telos NOR – 1 MB	0.368	Telos	write+read
Micron NAND – 128MB	0.017	Mica2	erase+write+read

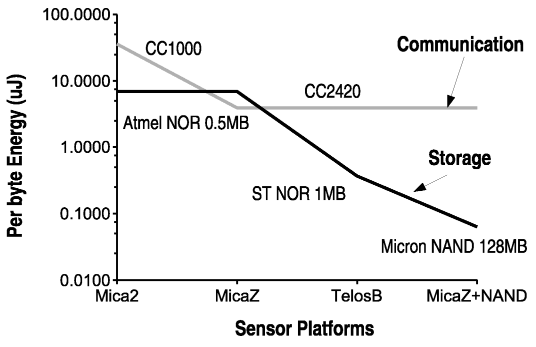

Fig. 1. Energy cost of storage compared to most energy-efficient radio (CC2420 radio) used on Mote platforms. Note that values include energy used by the CPU during device driver execution.

generations of flash memories not only offer higher capacity, but do so at decreasing energy costs per byte stored. Equipping the MicaZ platform with NAND flash memory allows storage to be *two orders of magnitude cheaper* than communication, and comparable in cost to computation. (i.e., only several times more expensive than copying data in RAM.) Figure 1 compares the per-byte energy cost of computation, communication and storage for various sensor platforms, and shows that the cost of storage has fallen drastically with the emergence of efficient NAND flash memories. This observation fundamentally alters the relative costs of communication versus computation and storage, making local archival on sensor platforms attractive as an alternative to communication, where it was not in the past.

Based on these trends, we argue that the design of sensor network architectures should include consideration of the 3-way trade-off among computation, communication, and storage, rather than ignoring storage as is often done today. This in turn results in the following recommendations: (i) Emphasis should be placed on in-network storage at sensors, and in particular most nodes should be equipped with high-capacity, energy-efficient local flash storage. (ii) Algorithms design should take into account the potential for cheap storage to reduce expensive communication. (iii) There is a need for an energy-efficient storage solution that allows sensors to maximally exploit storage for in-network data storage and archival; Capsule is an example of such a system.

1.3 Research Contributions

In this article we survey existing storage systems for sensor network systems, describe their shortcomings for sensor applications, and propose *Capsule*, which overcomes these drawbacks. We then present the design of the Capsule system, and give analysis and experimental results demonstrating its advantages. Our design and implementation provides the following contributions over prior approaches:

Object-Based Abstraction. Capsule provides the abstraction of typed storage objects to applications; supported object types include streams, indexes, stacks and queues. A novel aspect of Capsule is that it allows composition of objects—for instance, a stream and index object can be composed to construct a sensor database, while a file object can be composed using buffers and a multilevel index object. In addition to allowing reads and writes, objects expose a data structure-like interface, allowing applications to easily manipulate them. Storing objects on flash enables flexible use of storage resources: for instance, data-centric indexing using indices, temporary buffers using arrays, buffering of outgoing network packets using queues and storing time-series sensor observation using streams. Furthermore, the supported objects can also be used by applications to store live data and thereby use flash as an extension of RAM.

Portability Across Flash Devices. Capsule has been designed assuming only the common characteristics of both NAND and NOR flash memories. It employs a flash abstraction layer that uses a log-structured design to hide the low-level details of flash hardware from applications, allowing Capsule to function on any flash memory.

Energy-Efficient and Memory-Efficient Design. While traditional storage systems are optimized for throughput and latency, Capsule is explicitly designed for energy- and memory-constrained platforms. Capsule achieves a combination of very high energy-efficiency and a low memory footprint using three techniques: (a) a log-structured design along with write caching for efficiency, (b) optimizing the organization of storage objects to the type of access methods, and (c) efficient memory compaction techniques for objects. While its log-structured design makes Capsule easy to support on virtually any flash storage media, this paper focuses on exploiting the energy efficiency of NAND flash memories.

Handling Failures Using Checkpointing. Sensor devices are notoriously prone to failures due to software bugs, system crashes, as well as hardware faults due to harsh deployment conditions. Capsule simplifies failure recovery in sensor applications by supporting checkpoints and rollback—it provides energy-efficient support for checkpointing the state of storage objects and the ability to rollback to a previous checkpoint in case of a software fault or a crash.

To evaluate the effectiveness of the Capsule design, we have augmented Mica2 Motes with a custom-built board allowing the use of external NAND flash, and have implemented Capsule in TinyOS with an option to use either the Mica2 NOR flash memory or our custom NAND flash board. We perform a detailed experimental evaluation of Capsule on our storage-centric camera sensor network to demonstrate its energy efficiency. Our results show that even after extensively using Capsule to store and process camera images, Capsule consumed only 5.2% of the total energy consumed by the system. In addition, we have compared our file system implementation against Matchbox, and conclude that not only does Capsule provide a significantly more useful set of features, but does so with better performance and a better overall energy profile as well.

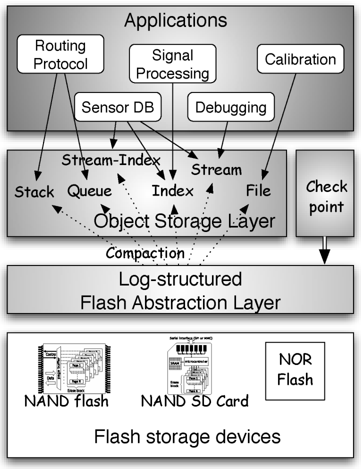

Fig. 2. Object Storage architecture.

The remainder of this article presents an overview of the Capsule architecture in Section 2, discusses its design in Sections 3 and 4, and presents Capsule implementation and evaluation in Sections 5 and 6. Section 7 shows the use of Capsule in a storage-centric camera sensor network, while we discuss related work in Section 8 and conclude in Section 9.

2. CAPSULE ARCHITECTURE

Capsule employs a three layer architecture consisting of a flash abstraction layer (FAL), an object layer, and an application layer (see Figure 2). The FAL hides the low-level flash hardware details from the rest of the object store. It provides an explicitly log-structured store of variable length records or *chunks*. Chunks are buffered as they are received from the object layer and written in batches, possibly interleaving chunks from multiple objects. At read time, however, chunks may be addressed and retrieved individually. Writes do not overwrite old data, but rather allocate new storage; when storage runs low, the FAL triggers a reclamation or *cleaning* process which explicitly garbage-collects stale data.

The object layer resides above the FAL. This layer provides native and flash-optimized implementation of basic objects such as streams, queues, stack and static indices, and composite objects such as stream-index and file. Each of these structures is a named, persistent object in the storage layer. Applications or higher layers of the stack can transparently create, access, and manipulate any supported object without dealing with the underlying storage device.

In addition to storage and retrieval methods for use by the application, each object class in Capsule supports efficient compaction methods that are invoked by the FAL when a cleaning operation is triggered. Finally, the object layer supports a checkpointing and rollback mechanism to enable reliable recovery of object data from software faults or crashes.

2.1 Flash Abstraction Layer

The Capsule FAL is shaped by the constraints of the hardware on which it may be implemented, and in particular those of the targeted flash devices. These are block storage devices, organized into *pages* of typically 512 or 2048 bytes; however, unlike a disk drive it is possible to read (or sometimes write) partial portions of a page. A key constraint of all flash devices is that data cannot be rewritten—once written, a location must be reset or *erased* before it may be written again. These erase operations are relatively slow and expensive, and must be performed in granularity of an *erase block*, which usually spans multiple pages, complicating storage management.

High-density NAND flash devices impose additional constraints: the number of nonoverlapping write operations in each page is typically limited to between 1 and 4, and (for some devices) pages within an erase block must be be written sequentially. Finally, flash devices will degrade after a certain number of write cycles—typically 10^5—and so care must be taken to spread writes across the device rather than repeatedly erasing and rewriting the same location.

To support flash portability in Capsule, we must conform to the strictest constraints placed by any flash memory technology we wish to support. Since NAND flash constraints are the most restrictive, we use these in Capsule: (i) Data may not be overwritten until it is erased; (ii) writes should be mapped to pages in a way that spreads them across all pages, to avoid degradation, and (iii) writes must be ordered sequentially within each page and within each erase block.

These constraints lead directly to a log-structured approach, where new information is written sequentially to storage as it arrives, rather than in predetermined locations. Although this approach was originally applied to storage systems (in Rosenblum and Ousterhout's Sprite LFS [Rosenblum and Ousterhout 1992]) for the purpose of minimizing disk seeks when writing, it is applicable to a broader range of problems, as we see here. The FAL treats the storage device as a "log"—it sequentially traverses the device from start to the end, writing data to consecutive pages. Once data has been written to a segment of the storage device it cannot be modified until it has been freed and erased.

By buffering small writes and aggregating them into full-page write operations we eliminate the need to read existing data and then rewrite it, thus minimizing the energy required. By moving the write frontier sequentially, out-of-order writes to pages within an erase block are avoided. In addition, since a page will not be rewritten until the write frontier wraps around and all other pages have been written once, we maximize the time until any single page reaches its write cycle limit—in order for any page to reach 10^5 write/erase cycles, a total of nearly $N \cdot 10^5$ pages must be written, where N is the size of the flash.[1] At the same time, the ability of flash devices to quickly and efficiently perform small, randomly-addressed reads allows us to easily retrieve data which has been stored in an interleaved log. By taking this log-structured

[1]Given proper wear leveling, we note that energy limits will prevent small wireless sensors from exceeding write cycle limits. For instance, at $0.017\mu J$ per byte, 10^5 write cycles on a 1GB device would require $6 \cdot 10^5 J$, as compared to the capacity of 2 AA cells, $2 \cdot 10^4 J$.

Table III. Taxonomy of Applications and Storage Objects

Application	*Data Type*	*Storage object*
Archival Storage	Raw Sensor Data	Stream
Archival Storage and Querying	Index	Stream-Index
Signal Processing or Aggregation	Temporary array	Index
Network Routing	Packet Buffer	Queue/Stack
Debugging logs	Time-series logs	Stream-Index
Calibration	Tables	File

approach we are thus able to work within the hardware constraints imposed by NAND flash memories, and in turn meet the less restrictive constraints of NOR flash (e.g., on existing Mote platforms) with minimum modification.

In a log-structured system, modifications do not overwrite existing storage, but rather allocate new storage locations. As data is modified, this process will eventually fill all available storage with stale data, requiring some sort of *cleaner* task akin to a memory garbage collector. Cleaning algorithms such as hole-plugging [Matthews et al. 1997; Rosenblum and Ousterhout 1992] proposed for disk-based log-structured file systems, however, are not feasible on flash devices which lack in-place modification capability. Instead, the Capsule approach divides responsibility for compaction. The FAL is responsible for tracking flash usage, and under low storage conditions will notify the object layer that compaction is needed. Each object class, in turn, is responsible for traversing and compacting its own instances.

2.2 Object Storage Layer

While the Capsule FAL is largely shaped by system constraints, the organization of the Object Storage Layer is primarily determined by application requirements. A cross section of representative sensor network applications or use cases is shown in Table III, along with the corresponding Capsule object which has been defined to meet the needs of that application; these use cases are described below.

First, many data-centric applications and research efforts need the capability to perform in-network archival storage, indexing and querying of stored data. Sensor data is typically stored as time series streams that are indexed by time [Li et al. 2006], value [Zeinalipour-Yazti et al. 2005], or event [Ratnasamy et al. 2002]. Such applications need to efficiently store *data streams* and maintain *indices*.

A second class of applications that can take advantage of efficient storage are those that need to use flash memory as a backing store to perform memory-intensive computation. Many data-rich sensing applications such as vehicle monitoring, acoustic sensing, or seismic sensing need to use large *arrays* to perform sophisticated signal processing operations such as FFT, wavelet transforms, etc. In cases where such processing is only needed infrequently, external storage may allow the use of a much smaller, cheaper, and less energy-intensive processor for the task.

A number of system components can also benefit from efficient external storage. The radio stack in TinyOS [Levis et al. 2005] does not currently support packet queuing due to memory limitations on supported CPUs; *Queue* objects could be used to buffer packets on flash in an energy-efficient manner. Debugging distributed sensors is often a necessary aspect of sensor network deployments, and requires efficient methods for storing and retrieving debugging *logs* [Ramanathan et al. 2005]. Other uses of the object store include support for persistent storage of calibration *tables*, corresponding to different sensors on the node. Finally, there is a need to support a *file system* abstraction to easily migrate applications that have already been built using existing sensor file systems such as Matchbox.

Based on this taxonomy of flash memory needs of applications (Table III), we identify a set of objects—Stream, Queue, Stream, Index, Stream-Index, and File—that correspond to these application needs. Four of these (Stack, Queue, Stream, and Index) form the basic first-order objects in Capsule, while Stream-Index and File may be composed from multiple basic objects.

3. FLASH ABSTRACTION LAYER DESIGN

Meeting flash device constraints was necessary in order for the Flash Translation Layer to function on the devices we targeted. In order to function well, however, it was necessary to optimize the FAL for energy efficiency and a low memory footprint. In the following section we provide details of this optimization, describing the trade-off between memory and energy for our preferred platform, NAND flash. In addition, we briefly discuss handling of flash hardware errors, as well as provisions for supporting non-Capsule-aware applications.

3.1 Energy / Memory Trade-Off

The goal of the Capsule system is to use as little memory as possible on resource-constrained sensor nodes, while at the same time minimizing energy use for typical sensor network applications. These criteria rule out existing NAND flash-based storage systems designed for portable devices, which have either large memory footprints, poor energy efficiency, or both. Some systems such as JFFS [Woodhouse 2001], for instance, maintain an in-memory logical-to-physical block map to simplify erase management, while others such as YAFFS [Manning 2002] maintain an in-memory map of the file blocks—in both cases using large amounts of memory. Energy consumption of these systems is high, as well, as even the smallest modification requires a read-modify-write of an entire page.

In order to minimize the amount of data that must be written to or retrieved from flash, the FAL supports read/write of variable-length records called *chunks*, each consisting of a 2-byte length, 1 byte of checksum, and a data field. Our design questions, then, boil down to these: When should chunks be written to flash? Where should they be stored until they are written? Similarly, when should they be retrieved? And where should retrieved chunks be stored?

In order to answer these questions we first look at our design constraints. Memory is the easy constraint—low-end sensors have between 4096 and 10240 bytes of RAM, of which a substantial fraction will no doubt be used by the application itself. Under the most demanding conditions it is unlikely that we

Table IV. Measured Energy Use and Latency of Flash Operations for Toshiba 128MB Flash

		Write cost (μJ)	Read cost (μJ)	Write latency	Read latency
NAND Flash (device only)	per operation	13.2μJ	1.073μJ	238us	32 us
	per byte	0.0202μJ	0.0322μJ	1.530us	1.761us
NAND Flash + CPU (driver execution)	per operation	24.54μJ	4.07μJ	274us	69us
	per byte	0.0962μJ	0.105μJ	1.577us	1.759us

would be able to reserve more than 500 or 1000 bytes for the storage subsystem, although in other cases somewhat more memory might be available. Energy constraints are more difficult to quantify, as the energy used is dependent on the flash access pattern. In Table IV we see measured device and system energy costs incurred by the read and write operations of a Toshiba TC58DVG02A1FT00 128 MB NAND flash [Toshiba 2003]. This device, like the others supported by Capsule, supports variable-length reads and writes at non-block-aligned offsets. (subject to constraints discussed previously)

Our measurements show that energy use may be modeled accurately as a fixed cost per operation, plus an additional cost for each byte retrieved or written. In particular, device and system energy cost for a set of operations is:

$$\begin{aligned} E_{device} &= 13.2 \cdot ops_{write} + 0.020 \cdot bytes_{write} + \\ &\quad 1.07 \cdot ops_{read} + 0.032 \cdot bytes_{read}\ \mu J \qquad (1) \\ E_{system} &= 24.4 \cdot ops_{write} + 0.096 \cdot bytes_{write} + \\ &\quad 4.07 \cdot ops_{read} + 0.105 \cdot bytes_{read}\ \mu J. \qquad (2) \end{aligned}$$

The device energy value reflects measurements of the energy used by the flash device itself, and represents a lower bound on the efficiency achievable unless a lower-power device is used. The system energy value, conversely, reflects energy used by the entire system, including that necessary to power the CPU while executing the flash device driver and transferring data. Use of a lower-power processor than the Atmel AVR used in these experiments, or more highly optimized drivers, might result in lower energy use.

Write Buffering. In Figure 3 we see three possible write buffering strategies for the FAL: no buffering, per-object buffers, and a single pooled buffer. The unbuffered strategy is expensive in terms of both energy and storage space. The high per-operation write energy usage causes a series of small writes to use many times more energy than if the same data were written in a single page-sized operation. In addition, if a flash with page size p can only support k (typically 4 or 8) writes to a page between erase operations, either a buffer of p/k bytes is needed, or after a series of k small writes it will be necessary to move to a fresh page, wasting any remaining storage in the current page.

The per-object buffering strategy in Figure 3(b) allows us to accumulate an entire page of data and then write it in a single operation. This would result in significantly lower energy usage, due to the decreased number of write operations; however, memory usage would be high. In TinyOS, for instance, this would prevent the dynamic creation of storage objects, as buffers would have

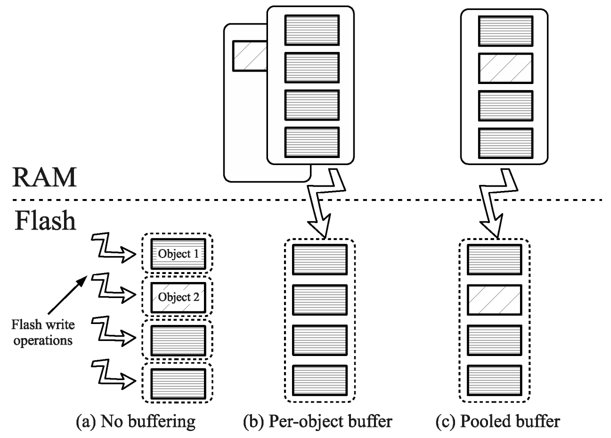

Fig. 3. Write buffering alternatives for the Capsule FAL.

to be allocated at compile time. Even worse, buffers might never be flushed to flash if (as we expect) typical object sizes are smaller than a full page.

Instead, we use the log-structured approach shown in Figure 3(c), where writes to different objects are interleaved in the order they are received, and stored in a single shared buffer. When this buffer fills, or optionally when the application forces a flush operation, the buffer is written to flash and then cleared. The size of this buffer may be configured at compile time; the minimum size is p/k as described before, but we advocate that *the* FAL write buffer be made as large as possible given system memory constraints, up to a full flash page.

Read Buffering. As shown in Table IV, read operations have very little per-operation overhead. In turn, this means that data transfer is expensive in comparison: reading an extra 30 or 40 bytes uses as much energy as invoking an additional read operation. We therefore avoid any form of speculative read operations (e.g., reading an entire block when a data chunk is requested), as these would not be justified unless we were very confident that the data would not be wasted. Since TinyOS is not a fully multitasking system we are unable to buffer multiple reads before they are issued; if we were able to it would still be unlikely that they would be contiguous.

For these reasons we avoid any form of read buffering in Capsule, instead transferring data directly into application memory. For fixed-length objects this is straightforward, as the buffer supplied in the read request can be expected to be the same length as the data to be retrieved. For variable-length objects, however, it is more complicated, as the application is likely to supply a buffer large enough to hold the maximum-length object. To optimize this case, our flash driver provides an efficient single-phase read mechanism for retrieving such chunks. As described above, each chunk contains a header with checksum and length. The length field is examined as it is received, and is then used to terminate the driver operation after the correct number of data bytes have been retrieved.

3.2 Storage Compaction

As described in Section 2, the FAL and Object Layer cooperate to reclaim storage when necessary. The design of this compaction process is based on three principles: (i) the FAL determines *when*, while the Object Layer is responsible for *how*; (ii) compaction and deletion should be deferred as long as possible, and (iii) simple cleaning algorithms are best for simple devices.

The FAL hides the size of the flash device and the amount of free storage from the Object Layer, in part because device characteristics may make it difficult to calculate how much additional application data the device is currently able to store. Instead, the FAL tracks device utilization, and when it reaches a configured threshold (by default 50%), the FAL issues a compaction event to the Object Layer. (In the TinyOS implementation, each upper-layer object exports a Compaction event input, which is wired to the FAL compaction interface.)

In the current implementation, each Capsule object implements a simple copy-and-delete compaction procedure, where each data structure is traversed and rewritten to the current write frontier. Once all object classes have performed compaction, the FAL marks the older blocks for deletion. These blocks are not erased until needed for new storage, thus preserving their contents as long as possible, for example, for checkpoint and rollback purposes.

Performance of the compaction process is dependent on the amount of live data to be copied vs. the size of the flash device. To derive the amortized cost of compaction, we analyze operation starting at the completion of one compaction operation until the end of the next compaction. If the fraction of storage used by live data is r, and the size of the device is N, then during this cycle we will write $N(1-r)$ bytes of data, followed by a compaction phase which copies Nr bytes of data; the fraction of bytes written by the reclamation process is thus r.

In order to guarantee enough free storage for the compaction phase to complete, r must be less than 1/2; however at this utilization level, Capsule will perform as much work compacting as it will writing application data. We recommend that device utilization be kept substantially lower than 50%—for example, 25% should be feasible on large devices with dynamic data structures.

3.3 Error Handling

Flash memory is vulnerable to a number of types of error, including single-bit errors in NAND flashes, and corrupted data due to transmission errors or crashes during writes. The FAL provides support for a simple checksum for each chunk, enabling the FAL and higher layers to check for errors after reading a chunk. In addition, for NAND flashes which are vulnerable to single-bit errors, the FAL provides a single-error-correction double-error-detection (SECDED) page level code. If the chunk checksum cannot be verified, the entire page is read into memory, and the error correction operation can be performed using the SECDED code. Error correction can be disabled for extremely memory limited sensor platforms, or if more reliable NOR flashes are used, as it it necessitates allocating an extra page-sized read buffer; our experience has shown these errors to be rare in practice. Our current implementation supports the chunk-level checksums, and support for page-level error correction is part of our future plans.

Table V.
Complexity Analysis of Storage Object Methods. Here N = Number of Elements in the Storage Object, H is the Number of Levels in the Index, and k = Number of Pointers Batched in a Chunk for Compaction or Indexing

Object Name	Operation	Energy Cost
Stack	Push	1 chunk write
	Pop	1 chunk read
	Compaction	N header reads, chunk reads and chunk writes + $\frac{N}{k}$ chunk reads and writes
Queue	Enqueue	1 chunk write
	Dequeue	N-1 chunk header reads + 1 chunk read
	Compaction	Same as Stack
Stream	Append	1 chunk write
	Pointer Seek	0
	Seek	N-1 chunk header reads
	Next Traversal	N-1 chunk header reads + 1 chunk read
	Previous Traversal	1 chunk read
	Compaction	Same as Stack
Index	Set	H chunk write
	Get	H chunk read
	Compaction	$\frac{k^H-1}{k-1}$ chunk reads and writes

3.4 Block Allocation

The FAL also offers a raw read and write interface that bypasses the log-structured component and accesses the flash directly. The FAL designates a part of the flash (a static group of erase blocks) for special objects or applications that directly access the flash. Such direct access is necessary for *root directory* management performed by the Checkpoint component (discussed in Section 4.4), which is used for logging critical data and needs to have control over when this data is committed to storage. In addition to checkpointing, network re-programming (e.g., Deluge [Hui and Culler 2004]) requires direct flash access, as to re-program a Mote reprogramming module needs to store new code contiguously in flash, without FAL headers.

4. OBJECT STORAGE LAYER DESIGN AND ANALYSIS

The Capsule Object Storage layer resides above the FAL, and provides both basic and composite objects based on our application taxonomy described above. Basic objects are based on linked lists (Stream, Queue, and Stack) and a k-ary tree for the Index. Composite objects are the Stream/Index, that provides powerful methods for retrieving data by value or by temporal position, and a file system that provides a flexible and powerful replacement for systems such as Matchbox. Each of these objects is designed for simplicity, using only small amounts of RAM buffering, code space, and CPU cycles.

In this section we discuss these storage objects and their methods (summarized in Table V) in more detail. We describe for each their internal implementation and external access methods supported, and analyze the energy costs of both access and compaction operations. In addition, we describe how checkpointing and rollback is supported in our system for failure recovery.

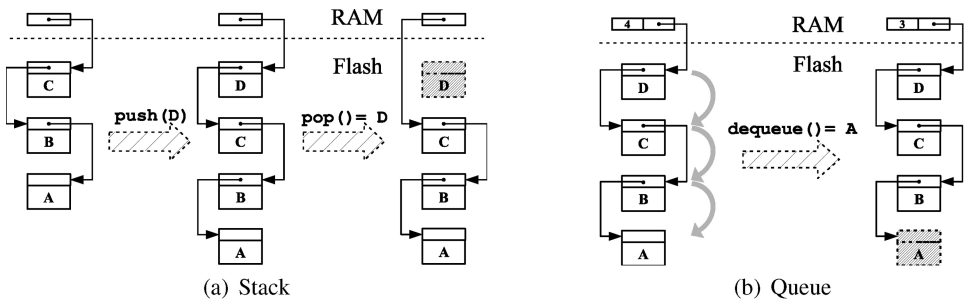

Fig. 4. Structure and operation of the Queue and Stack objects.

4.1 Linked List-Based Objects

Three of the basic objects identified in our taxonomy in Table V are implemented using a simple linked list structure, where each record written to flash contains a header with a pointer to the previously written record, and a list head pointer is maintained in RAM. The Queue object is the simplest of these—`push()` writes a new record and updates the head pointer, while `pop()` retrieves the most recent record, the address of which may be found in the head pointer in RAM. The link pointer in the retrieved record will indicate the previous record, and will be used to update the RAM head pointer. The Queue and Stack operations are shown in Figure 4.

The Queue object implementation is more complex, as the lack of in-place update forces us to use the same reverse-linked list as a queue. To retrieve the record at the tail of the queue it is necessary to traverse the entire list, at a cost of retrieving N record headers where N is the length of the queue.[2] In addition the in-RAM state for a queue must include a length count so that we can locate the end of the queue, as we are unable to modify link pointers when removing items from the tail of the queue. The Stream object, in turn, is similar to the Queue, with additional methods for seeking to absolute locations within the list, as well as traversing in either direction from a given point.

Compaction. As the Queue, Stack, and Stream objects use the same in-flash data structure, the compaction process is identical for each. It involves retrieving each object, and then rewriting them in the order that they were originally written. To avoid traversing the list for each object, we use a two-step scheme, as shown in Figure 5. First, the list is traversed from head to tail (last inserted to first inserted element) at cost $N \cdot R(h)$ (see Equation (2)), where N is the number of elements in the stack and h is the total header size, that is, the sum of FAL and stack headers. The pointers for each of these elements are written into a temporary stack of pointers, perhaps after batching k pointers together in each write incurring cost $\frac{N}{k} \cdot W(d)$, where d is the size of a stack chunk. Next, the stack of pointers is traversed (at cost $\frac{N}{k} \cdot R(d)$) and each data chunk corresponding to the pointer is now read and then rewritten to the FAL to create the

[2]The inability to modify data after it is written to flash rules out the traditional queue implementation where new items are linked to the object at the tail.

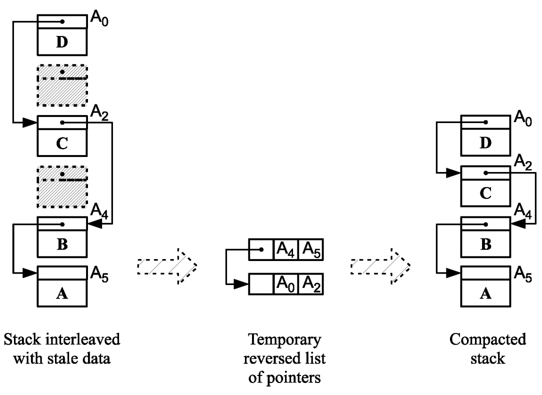
A0
D
A2
C
A4
B
A5
A
A4 A5
A0 A2
Stack interleaved with stale data
Temporary reversed list of pointers
Compacted stack

Fig. 5. Compaction of the Stack object.

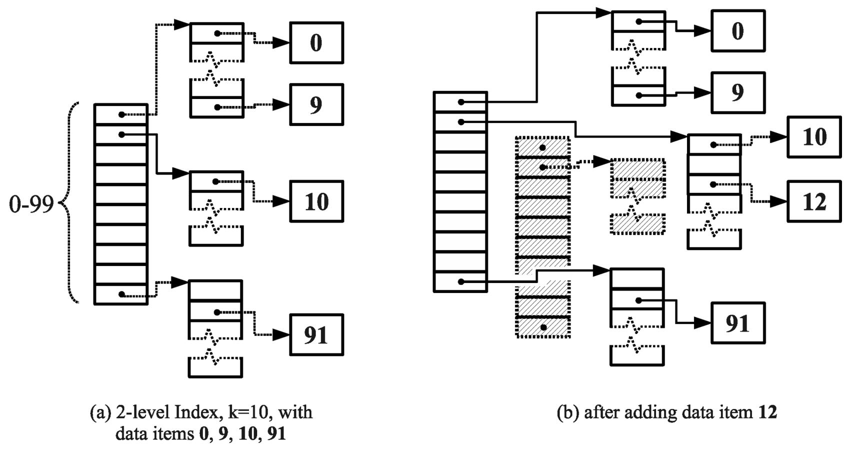
0-99
0
9
10
91
12
(a) 2-level Index, k=10, with data items **0, 9, 10, 91**
(b) after adding data item **12**

Fig. 6. Organization and compaction of the Static Index object.

new compacted stack (at cost $N \cdot (R(d) + W(d))$). The total cost is, therefore,

$$(N + \frac{N}{k}) \cdot (R(d) + W(d)) + N \cdot R(h),$$

as shown in Table V. Note that the reclamation cost is independent of the amount of stale storage which is reclaimed. This is the case with other object types, as well, due to the use of a copying compaction mechanism.

4.2 Static-Sized Index/Array

Our trie-based index object permits data to be stored using $(key, opaque\ data)$ format, and supports a range of key values that is fixed at compilation time. Since this object provides an access pattern similar to that of an array, we use both interchangeably in this paper. Figure 6 shows the structure of the index object—it is hierarchical with a fixed branching factor k at each level ($k = 10$ in the figure). Due to platform limitations (i.e., TinyOS does not support dynamic memory allocation) the number of levels for any individual object is fixed.

Figure 6 shows the construction of a two-level index. Level zero of the implementation is the actual *opaque data* that has been stored. Level one of the index points to the data, and level 2 of the index aggregates the first level nodes of the index. Each level of the index has a single buffer that all nodes at that level share. For example, the `set` operation on the index looks up and then loads the appropriate first level index corresponding to the *key*. It then writes the *value* to FAL and updates the first level of the index with the location of the written *value*. If the next *set* or *get* operation operates on the same first level index node, then this index gets updated in memory. But if the next operation requires some other first level index node, the current page is first flushed to flash (if it has been modified) and then the second level index is updated similarly, and finally the relevant first level page loaded into memory.

Compaction. This involves traversing the object in a depth-first manner, reading the pointers and writing them to FAL. The cost of the compaction operation is therefore the same as the cost of reading the index in a depth-first manner, and writing the entire index into a new location in flash. If the index has H levels, and each index chunk can store k pointers, the total number of index chunks in the tree is $\frac{k^H-1}{k-1}$. Compaction involves reading and writing every chunk in the index, and has cost

$$\frac{k^H-1}{k-1}(R(d)+W(d)).$$

4.3 Composite Storage Objects

The object store permits the construction of composite storage objects from the basic set of objects that we described. The creation and access methods for the composite objects are simple extensions of the primary objects, but compaction is more complex and cannot be achieved by performing compaction on the individual objects. Object composition is currently done "by-hand" in Capsule; making nesting of objects simpler is part of our future plans. We present two composite storage objects. The first composite object that we describe is a *stream-index* object that can be used for indexing a stored stream. For instance, an application can use a stream-index to store sensed data and tag segments of the stored stream where events were detected. Second, we describe our implementation of a file system in Capsule using a *file* composite storage object that emulates the behavior of a regular file. This object facilitates porting applications that have already been developed for the Matchbox filesystem [Gay 2003] to Capsule.

4.3.1 *Stream-Index.*

The stream-index object encapsulates a stream and an index object and offers a powerful interface to the application. The application can directly archive its data to this object by using the `add` method, which saves the data to the stream. When an event is detected in the sensed data, it can be tagged using the `setTag` method, which stores the pointer to the stored stream data in the next free *key* in the index object. This interface can also be trivially modified to tag ranges of sensor readings instead of a single reading. The `seek` method allows the application to seek into the stream based on a given

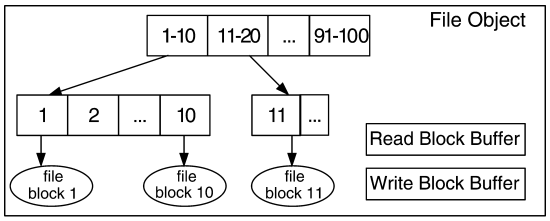

Fig. 7. Design of a filesystem using Capsule.

tag, and the `next` and `previous` methods allow the application to traverse data in either direction.

4.3.2 *File System.* Capsule can be used to construct a simple filesystem with the help of the index object discussed earlier—Figure 7 shows our design and we provide a brief overview of our implementation. A file system object is composed of two classes of objects—*file* objects, representing individual files, and a singleton *file-system* object, storing metadata associated with each file. The file system object is responsible for assigning each file a unique *file-id* and mapping the filename to its associated file-id. Each file-id maps to a unique file object, which is actually a static index storing the contents of the file in separate fixed-size *file blocks*, with the index pointing to the location of each file block on flash. The object also maintains the current length of the file (in bytes) and any working read/write pointers that the file may have. Additionally, associated with each open file are two file block-sized buffers, serving as read and write caches. When performing a data write, the data is copied into the write cache, which is flushed when filled. Similarly, data is first read into the read cache before being returned to the application. Organizing the file blocks using the index object allows us to support *random access* to each block. We use this feature to modify previously written data, by first loading the appropriate file block from flash, modifying the relevant bytes, writing the block to a new location, and then updating the index accordingly. The previous block is now no longer referenced, and will be reclaimed after compaction.

Our implementation supports simultaneous reads and writes to a single file, and multiple files can be open and operated upon at the same time. In addition, our file system takes advantage of the checkpoint-rollback capability of Capsule to provide consistency guarantees. These features are not supported by ELF and Matchbox, and demonstrate the flexibility of object composition within Capsule. In addition to this comparison of features, in Section 6.3 we provide a performance comparison Capsule and Matchbox, as well.

4.4 Checkpointing and Rollback

Capsule also supports capability for checkpointing and rollback of objects: checkpointing allows the sensor to capture the state of the storage objects, while rollback allows the sensor to go back to a previously checkpointed state. This not only simplifies data management, but also helps recover from software bugs, hardware glitches, energy depletion, and other faults which may occur in sensor nodes.

The inability of flash to overwrite data once written in fact simplifies the implementation of checkpointing. The internal pointers of an object (e.g., the next pointer for a stack or a queue) cannot be modified once they are written to flash. The in-memory state of a storage object (which typically points to its written data on flash) thus becomes sufficient to provide a consistent *snapshot* of the object at any instant. The in-memory states of all active storage objects, then, provides a snapshot of the entire system at any given instant. We implement checkpointing support using a special *checkpoint* component, which exposes two operations—`checkpoint` and `rollback`. The `checkpoint` operation captures the snapshot of the system and stores it to flash. This saved snapshot can be used to revert to a consistent state in the instance of a system failure or object corruption.

The Capsule storage objects implement a `serialize` interface. The checkpoint component calls the `checkpoint` method on this interface when it needs to take a snapshot. This method is passed a shared memory buffer where it stores its in-memory state, which is then written to flash. The checkpoint component uses a few erase blocks in flash as the *root directory* that it manages explicitly, bypassing the FAL (Section 3.4). Once the checkpoint data has been written to flash, a new entry is made to the root directory pointing to the created checkpoint.

In the event of node restart or an explicit rollback call from the application, the root directory is searched to find the most recent checkpoint, which is used to restore system state. CRCs are maintained over the checkpoint data and the root directory entries to prevent corrupt checkpoints (possibly caused by the node crashing while a checkpoint is being created) from being recovered. The root directory entry provides a pointer to the saved checkpoint state, and the the checkpoint component uses the `rollback` method in the serialize interface to replace the in-memory state of linked objects using the same shared buffer mechanism as `checkpoint`.

5. IMPLEMENTATION

Implementing Capsule[3] presented a number of unique challenges. TinyOS is event-driven, and thus Capsule is written as a state machine, using the split-phase paradigm. The checkpointing component required careful timing and co-ordination between components for its correct operation. We went through multiple iterations of Capsule design to maximize code and object reuse even within our implementation—for example, the checkpoint component uses a stack object to store state information, as do the stack compaction methods for streams and stacks (Section 4.1). Another major concern was the overall memory footprint of Capsule. We optimized the Capsule architecture to minimize buffering and maximize code reuse; buffers have only been used at stages where they have a sufficient impact on the energy efficiency. A test application that does not use checkpointing/recovery but uses one instance of each of the following objects—index, stream, stream-index, stack, and queue, requires only 25.4Kb of ROM and 1.6Kb of RAM. Another application that uses

[3]Capsule source code is available at `http://sensors.cs.umass.edu/projects/capsule/`

one each of the stack and stream objects along with checkpointing support, had a foot-print of 16.6Kb in ROM and 1.4Kb in RAM. While the Capsule code base is approximately 9000 lines of code, the percentage of the code used by an application depends largely on the number and type of objects instantiated and the precise Capsule features used.

6. EVALUATION

In this section, we evaluate the performance of Capsule on the Mica2 platform. While Capsule works on the Mica2 and Mica2dot NOR flash as well as our custom NAND flash adapter, the energy efficiency of the NAND flash [Mathur et al. 2006a] motivated its use as the primary storage substrate for our experiments. However, the experiment comparing the Capsule and Matchbox file systems is based on the Mica2 NOR flash (Section 6.3), demonstrating Capsule's portability.

Our evaluation has four parts—first, we benchmark the performance of the FAL, including the impact of read and write caching. Second, we perform an evaluation of the performance of the different storage objects, measuring the relative efficiency of their access methods, and the impact of access pattern and chunk size on the performance of checkpointing and storage compaction. Third, we describe interesting trade-offs that emerge in an application study that combines the different pieces of our system and evaluates system performance as a whole. Finally we compare the performance of a file system built using Capsule (Section 4.3.2) with Matchbox. No such comparison was possible between Capsule and MicroHash, which relies on the significantly greater memory size of the RISE platform [Mitra et al. 2005], and could not be used on the platforms supporting Capsule.

Experimental Setup. We use our fabricated NAND flash adapter for the Mica2 with the Toshiba 128 MB flash [Toshiba 2003] for our experiments; the device has a page size of 512 bytes, an erase block size of 32 pages and permits a maximum of 4 nonoverlapping writes within each page. Our measurements involved measuring the current at the sensor and flash device power leads, with the help of a 10Ω sense resistor and a digital oscilloscope. The mote was powered by an external power supply with a supply voltage of 3.3V; energy consumption "in the field" with a partially discharged battery may be somewhat lower.

6.1 FAL Performance

The choices made at the FAL are fundamental to the energy usage of Capsule. We ask two questions in this section: How much write buffering should be performed at the FAL layer? and How much buffering should be performed by a higher layer before writing to FAL? To answer these, we vary write and read buffer sizes and examine the energy consumption of the write and read flash operations. Figure 8 shows our results, where each point corresponds to the energy consumed by writing or reading one byte of data amortized over a buffer of that particular size.

Impact of FAL Write Buffer Size. For this particular flash the minimum write buffer size is 128 bytes, to avoid exceeding the limit of 4 writes per 512 byte

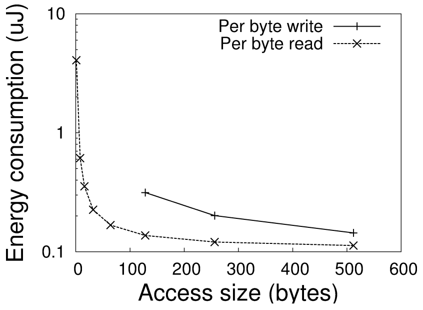

Fig. 8. The amortized energy consumption of the read and write operations was measured for different data sizes using the Mica2 and the fabricated Toshiba 128 MB NAND adapter. The figure clearly shows that the write operation has a high fixed cost involved in comparison to the read operation.

page, as described in Section 3. The per-byte write curve in Figure 8 shows that write costs increase significantly as write buffering decreases. A reduction in buffer size from 512 bytes to 128 bytes saves 9.4% of the available memory on the Mica2, but the per byte write energy consumption increases from 0.144μJ to 0.314μJ, that is, 118%. Reducing memory consumption from 512 bytes to 256 bytes results in memory savings of 6.3% of the available memory on the Mica2 mote, but at 40% additional energy cost.

Thus, increased write buffering at the FAL has a considerable impact on reducing the energy consumption of flash write operations—consequently, the FAL write buffer should be a full page, or if not, then as large as possible.

Higher Layer Buffer size. In our log-structured design, chunks are passed from the object layer to FAL, and are not guaranteed to be stored contiguously on flash. As a result, reads of consecutive chunks must be performed one at a time, since consecutive object chunks are not necessarily spatially adjacent on flash. To amortize the read cost, data buffering needs to be performed at the object or application layer. We aim to find the appropriate size of the higher layer buffer through this experiment.

Figure 8 illustrates the effect of chunk size on the cost of flash reads. Similar to write costs, the per-byte cost of a read reduces with increasing buffer sizes, dropping sharply (by 72%) as the buffer size increases from 8 bytes to 64 bytes. However, beyond 64 bytes the per byte cost decreases more slowly, and larger read buffers have relatively less impact. For example, increasing the write buffer from 128 bytes to 512 bytes results in a gain of 0.17μJ, whereas the same increase in read buffer size results in a gain of only 0.024 μJ, that is, only 14% of the write benefit.

Thus, approximately 64 bytes of data buffering at the storage object or application layer is sufficient to obtain good energy efficiency for flash read operations.

6.2 Performance of Basic Storage Objects

In this section, we first evaluate the energy efficiency of each access method supported by the core Capsule objects. Then, we present some important

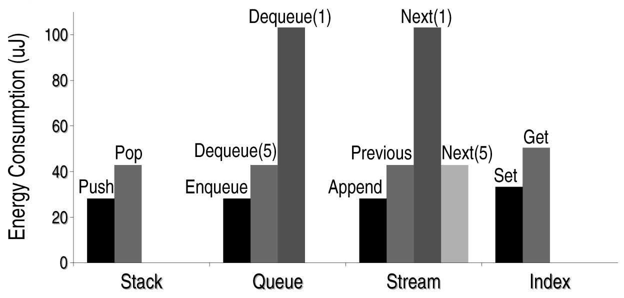

Fig. 9. Breakdown of energy consumed by the operations supported by core Capsule storage objects storing 64 bytes of data. The operations resulting in writes to FAL consume substantially less energy than the read operations.

trade-offs that arise in the choice of chunk size based on the access pattern of an object, using the Index object as a case study. Finally, we measure the performance of compaction and checkpointing.

6.2.1 *Energy Consumption of Object Operations.* Table V presents an analysis of the energy consumption of the access methods supported by the core objects. In this experiment we use micro-benchmarks to evaluate energy consumption of methods on different objects. The following specific choice of operating parameters is used.

—*Stack, Queue*, *and Stream Objects*. Each object stores 5 elements, each of size 64 bytes.

—*Index Object*. A two-level index, where the second level is cached in memory, and the first level is read from and written to flash as discussed in Section 4.2. Each index node holds 5 pointers to the next lower level.

A 512-byte FAL write buffer was used in all experiments. In each test 5 elements were stored in the object, and then the 5 elements were retrieved; in cases where energy consumption varied between calls to the same operation, the order of the operations is indicated. Thus `dequeue(1)` is the first `dequeue` operation and `dequeue(5)` the last.

Figure 9 provides an energy consumption breakdown of the Capsule object methods.

—*Stack*. the energy cost of the `push` operation and the `pop` operation are not affected by repeated operations, as each operates directly on the top of the queue. The `push` operation uses 34% less energy than that `pop`, as multiple `push` may be buffered in the FAL, while each `pop` results in a read to flash.

—*Queue*. the energy cost of the `enqueue` operation is the same as the Stack `push` operation, as the work done is equivalent. The cost for `dequeue` varies; the first operation executed (`dequeue(1)`), which must traverse pointers from the tail to the head, is 2.4 times more expensive than `dequeue(5)`, which removes the last remaining element and is equivalent to a Stack `pop`.

—*Stream*. the Stream object combines traversal methods of both the Stack and the Queue – the `append` operation is equivalent to `push`, `previous` to `pop`, and `next` to `dequeue`, with equivalent costs as seen in Figure 9.

—*Index*. The cost of `get` and `set` are constant across repetitions, as for Stack, but slightly higher due to the increased complexity of the structure. As with Stack, the read operation, `get`, is more expensive (by 52%) than the write operation, `set`, due to write buffering in the FAL.

For each of the object types, our measurements validate the cost analysis presented in Table V.

6.2.2 *Impact of Access Pattern and Chunk Size.* The access pattern of an object has considerable impact on the energy consumed for object creation and lookup, and we evaluate this in the context of the Index object (refer Section 4.2). We consider four different access patterns in this study: sequential and random insert, and sequential and random lookup. Our evaluation has two parts. First, we analytically determine the energy cost for each access pattern. Second, we quantify the cost for different combinations of insertion and lookup to identify the best choice of chunk size in each case. This study only considers the cost of indexing, and does not include the cost of storing or accessing the *opaque data* pointed to by the index.

Cost Analysis. We use the following terms for our analysis: the size of each index node is d, the number of pointers in each index node is k, and the height of the tree is H. The cost of writing d bytes of data is $W(d)$ and reading is $R(d)$ as per Equation (2).

Insertion into the Index object has two steps—first, H index chunks (root to leaf) are read from flash to memory, then insertion results in H index writes from the leaf up the root as we described in Section 4.2. Index lookup operations must read in the H index chunks corresponding to each level of the tree before retrieving the stored data. In each case the cost is different for sequential access, where data can be buffered or cached, and random access where it is not.

—*Sequential Insert*. index tree nodes can be cached, and are written only when the next element crosses the range supported by the node. One chunk write and chunk read is performed for each of the H levels for every k elements: $E = \frac{H}{k} \cdot (W(d) + R(d))$.

—*Random Insert*. If data is inserted randomly, each write results in a read followed by write of an index chunk at each level of the index: $E = H \cdot (W(d) + R(d))$.

—*Sequential Lookup*. As with writes, sequential reads cost less due to index node caching, and the amortized cost of sequential lookup is: $E = \frac{H}{k} \cdot R(d)$.

—*Random Lookup*. Random reads force each level of the index to be loaded afresh for each lookup operation. This increases the lookup cost: $E = H \cdot R(d)$.

Measurement-Driven Analysis. Figure 10 shows index insertion and lookup costs for varying chunk sizes for an index of height $H = 2$, based on our cost analysis and index measurements. In this analysis, we fix the number of elements inserted into the index at 32768 (i.e., $N = 32768$) and vary the size of each index node d, and thus the number of pointers, k, in each index node. Using the results derived above, we are thus able to examine the effect of chunk

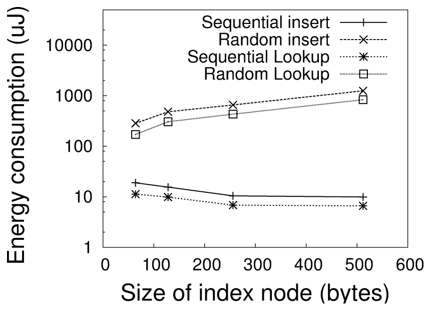

Fig. 10. Energy consumption of the index for varying insertion and lookup operational patterns, varying index node sizes—these are assume no associated data. Sequential insert and lookup are substantially more energy efficient than their random counterparts.

size on efficiency for applications with different expected insertion and lookup behaviors.

—*Random Insert—Random Lookup*. This access pattern results from a value-based index, where elements are inserted randomly and specific values are looked up. For this case, we seen in Figure 10 that energy usage is minimized when index nodes are the smallest possible—in this case 64 bytes or 15 elements per node, where each pointer is 4 bytes long.

—*Random Insert—Sequential Lookup*. This models the case where time-series data is stored in a value-based index, and then the entire index is read sequentially. (e.g., to build a histogram of the data) The choice of index node size depends on number of inserts as well as the number of lookups—insert is optimized at an index node size of 64 bytes, while optimum lookup performance occurs with a size of 256 bytes. If lookups are less frequent than inserts, then a small index node size should be used, while if lookups are frequent then a large size should be used.

—*Sequential Insert—Sequential Lookup*. An index maintaining time series data would store and later access the data sequentially. Larger chunk sizes result in better energy optimization, however, a buffer size of 256 bytes is sufficient as both insert and lookup costs are close to their lowest value.

—*Sequential Insert—Random Lookup*. An index maintaining time-series data would store data sequentially, but temporal queries on past data can result in random lookups. The optimal size again depends on the number of inserts and lookups—the insert is optimal at 64 bytes while the lookup become more efficient after 256 bytes. The ratio of the number of lookups to inserts would determine the choice of index size (similar to the random insert-sequential lookup case).

Our analysis shows that smaller index chunk sizes are favorable for random insertion and lookup operations, since smaller sizes lead to lower cost of flash read operations. Larger chunk sizes are better for sequential operations, since they utilize buffering better, resulting in greater in-memory updates and fewer flash operations.

6.2.3 *Storage Overhead.* Both the object layer and FAL add overhead: the object layer maintains an index with approximately 4 bytes of overhead for each data chunk when the index is full, while the FAL adds an additional 3 bytes per chunk. Matchbox, in contrast, has a fixed 8-byte overhead for each 256-byte block of data stored. The overhead comparison is thus dependent on the size of the application write operations and thus the chunks stored in the file. If these chunks are larger than 256 bytes, then Capsule will be more storage-efficient than Matchbox. In typical usage chunks would be smaller than this, but the amount of storage used by Capsule remains less than 10% for chunk sizes down to 36 bytes, while offering much higher levels of functionality.

6.2.4 *Storage Reclamation Performance.* Storage reclamation (Section 3.2) is triggered when flash usage reaches a predefined threshold. Our current implementation uses a simple compaction scheme, where the storage objects read all their valid data and re-write it to the current write frontier on the flash. To evaluate the performance of this approach, we present measurements of storage reclamation on the stream and index objects. We note that the compaction procedure and costs for stack and queue objects are identical to those of the stream object (Table V).

In our experimental setup, we trigger compaction when 128 KB of object data has been written to flash; our goal is to find the worst case time taken for compaction. In the case of the Stream object, an intermediate stack is used to maintain ordering of the elements post-compaction, as discussed in Section 4.1. For the 2-level Index object (discussed in Section 4.2), we set the second level of the index to hold 100 pointers to level 1 index nodes ($k_1 = 100$) and each level 1 node holds pointers to 50 data blobs ($k_2 = 50$). In the experiments, we vary the size of the data being stored in each object chunk from 32 bytes to 256 bytes, in order to measure the range of compaction costs. We first perform a measurement of the energy consumption of the compaction process followed by a measurement of the time taken.

Energy Consumption. Figure 11 shows the energy cost of compaction in comparison to the cost of sequential data insertion. We first consider the write and compaction costs for the Stream object—we observe that increasing chunk size reduces the cost of writing and compacting. The reduction in write costs is attributed to reduced header overhead from writing fewer chunks. The reduction in the stream compaction cost is considerably greater. As the size of data chunks increase, the number of elements in the stream decreases, which results in fewer pointer reads and writes to the intermediate stack during the compaction phase. Additionally, the efficiency of both read and write operations improves as the data size increases (refer Section 6.1). The compaction overhead can be reduced considerably by increasing the chunk size from 32 to 128 bytes—in fact, the savings equal about three times the cost of writing the original data. Further increase in chunk size results in smaller improvements in compaction performance.

The write and compaction costs for the Index object follow a similar overall trend. Interestingly, the write cost for the Index object is greater than that of the Stream object whereas the compaction cost of the Stream object is

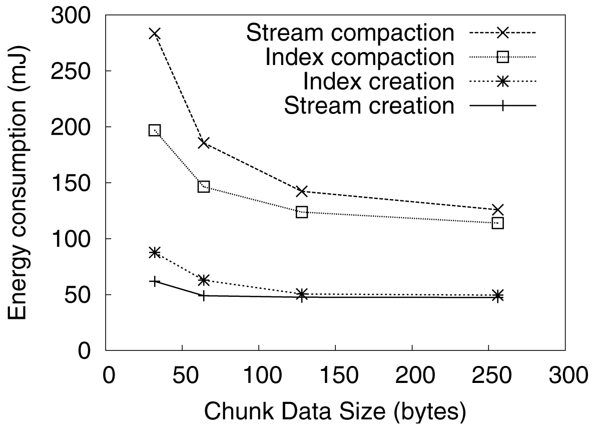

Fig. 11. The energy consumed by compaction not only depends on the amount of data, but also on the size of each data chunk of the object. The energy consumed by an Index and a Stream object holding 128KB of data is shown here for varying chunk data sizes. Larger object-level buffering requires fewer number of chunks to be read and written—the compaction costs more than double when changing buffering strategy from 32 bytes to 256 bytes.

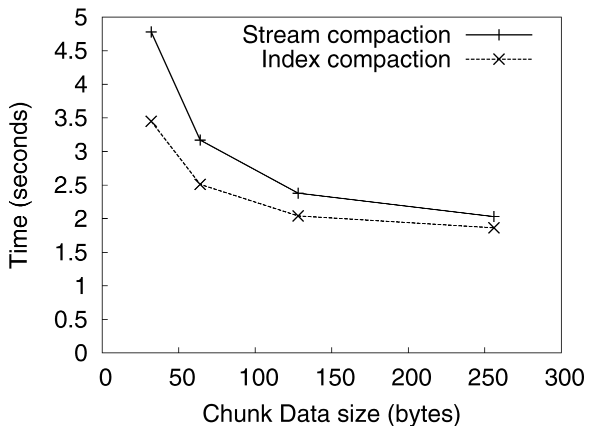

Fig. 12. The compaction time of the storage object is linked to both the amount of data the object holds and the size of each data chunk. The time taken to compact an Index and a Stream object holding 128KB of data is shown here for different data chunk sizes.

considerably higher than that for the Index object. This is because creating an Index object is more expensive due to the writing and reading of the level 1 index nodes. The compaction of the Index is less expensive than Stream compaction because Index compaction requires only a depth-first traversal of the index, while Stream compaction requires the creation and traversal of the intermediate pointer stack, which requires additional energy. If the fraction of discarded data is f for either the stream or the index, then the cost of compaction will be $(1 - f)$ times the corresponding point in Figure 11.

Latency. Figure 12 shows the latency of the compaction operation. This is also an important measure of compaction, as no operation can be performed

Table VI.
Energy Consumption and Latency of Performing Checkpointing Operations on a Stream and Index Object

Operation	Latency (μs)	Energy consumption (μJ)
Checkpoint	996	82.5
Rollback	284	42.1
Restore	460	50.87

on the object while it is being compacted. We find that in all cases the entire compaction operation executes in less than 5 seconds. This can be improved to 2.5 seconds for the Stream and to 2 seconds for the Index by increasing the data size to 128 bytes. This shows us that even while compacting 128K of object data, the storage object will be unavailable only for a short duration, and this can be dealt with easily by providing minimal application-level buffering.

The energy and latency results of compaction show that these operations can be performed efficiently on a small sensor platform. We find that a buffer size of 128 bytes provides a good balance between the memory needs of compaction and the energy consumption/latency of the process.

6.2.5 *Checkpointing.* Capsule supports checkpointing with the help of the special Checkpoint component that permits three operations: `checkpoint`, `rollback`, and `restore`. For our experiment, we consider a Stream and an Index object and link these to a single Checkpoint component. We then perform each of the operations permitted on the Checkpoint component and measure the latency of the operation and the energy consumed by the device—Table VI presents our results. We see that the latency of all the operations is less than 1 ms. The energy consumption of the `checkpoint` operation is approximately 3 times that of a stack `push` operation or only 2 times that of a `pop` operation with 64 bytes of data. The energy consumed by the `restore` operation is a little more than that of performing a `pop`, and the cost of `rollback` is equivalent to the cost of performing a `pop` operation on the stack. These measurements indicate that checkpointing support in Capsule is extremely low-cost and energy-efficient, allowing Capsule to support data consistency and crash recovery with minimal additional overhead.

6.3 Comparison with Matchbox

Having discussed the performance of the basic objects provided in Capsule, we evaluate how these objects can be used by applications and system components. We now compare our implementation of a file system based on Capsule (Section 4.3.2) with Matchbox [Gay 2003]. Our implementation also provides the following additional features: the ability to work with multiple files simultaneously, random access to a block in the file, modifying previously written data, and file consistency guarantees even in the event of system failure during a write operation.

Our experiment was performed on the Mica2 [Crossbow Technology 2004], using the platform's Atmel NOR flash. On both file systems, we created a new file and wrote 80 bytes of data in each of 10 consecutive operations, for a total of

Table VII.
Energy Consumption and Latency of Matchbox and Capsule Operations, Measured on Mica2 Mote. (Atmel AVR CPU, Atmel AT45DB041 NOR flash)

	Capsule		Matchbox	
	Energy (mJ)	Latency (ms)	Energy (mJ)	Latency (ms)
Create	1.79	19.16	1.03	14.16
Write (80b x 10)	8.83	85.6	10.57	91.60
Open	0.0093	0.184	0.093	1.384
Read (80b x 10)	1.20	18.440	1.12	16.520
Total (c+w,o+r)	11.83	123.4	12.82	123.7
Write Bandwidth	18.0kbps		11.3kbps	
Read Bandwidth	54.2kbps		60.4kbps	
Memory Footprint	1.5K RAM, 18.7K ROM		0.9Kb RAM, 20.1K ROM	

800 bytes. We then closed the file, reopened it, and read the 800 bytes similarly in 10 consecutive read operations of 80 bytes each. Table VII shows the performance of the Capsule file system in comparison to Matchbox. The memory footprint of both file systems is comparable; providing support for checkpointing as well as buffering at FAL, file and the index objects are the reason for the higher RAM footprint of Capsule. The individual energy consumption of file system operations on both is comparable. The write bandwidth provided by the Capsule file system is 59% more than Matchbox, while the read bandwidth lags by 10%. Considering the net energy consumption of the experiment, the Capsule file system turns out to be 8% more energy-efficient than Matchbox, while taking approximately the same amount of time.

Thus, our Capsule file system implementation provides rich additional features at an energy cost equivalent or less than that of Matchbox.

7. ARCHIVAL STORAGE AND INDEXING APPLICATION

As sensors are commonly used for archival storage and indexing, we measure and analyze the energy consumption of a storage-centric camera sensor network which archives and indexes its data. We then compare the energy usage of Capsule with that of the communication and sensing subsystems. Figure 13 shows our camera sensor network; each MicaZ sensing node is equipped with a Cyclops [Agilent] camera and our NAND flash adapter.

The sensor captures motion detection-triggered images that are *archived locally* on the NAND flash using Capsule. The images are stored in a *Stream* object, and an *Index* object is used to catalog the images by time. Smaller, subsampled versions of the stored images at each motion event are created and transmitted to the base station as an event summary; the reduction in size due to subsampling results in a corresponding reduction in transmit energy usage. At the base-station, which receives these subsampled images, the operator can examine them and determine if they represent events of interest—for example, a person walking by is an event of interest but a pet moving around is not. If the image or event is not of interest it can be discarded; however, for interesting events the full-resolution image can be requested from the sensor.

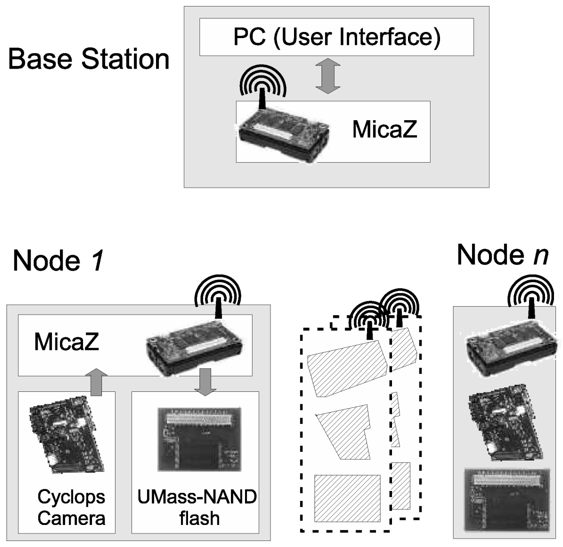

Fig. 13. Setup of the storage-centric camera sensor network.

By requesting images on an as-needed basis, extraneous images for object and event detection need not be transmitting on the power-hungry radio, resulting in energy savings. Additionally, the sensor uses wavelet transformation and run-length encoding (RLE) to reduce the size of the image that needs to be transmitted to the base-station, trading computation for more expensive communication. In spite of these operations, the size of the resulting image is much larger than the packet size, which is limited to 20 bytes. The sensor chops the image into packet-sized fragments and uses a flash-based packet queue to store each packet, which is then batch transmitted to the base-station.

We perform a component-level breakdown of the energy consumption of the storage, communication and sensing subsystems while performing these operations, and Figure 14 shows our results. We observe that Capsule consumes only 5.2% of the total energy while computation consumes 7.2%. The communication subsystem comes in third, consuming 30.6% of the net energy with the Cyclops camera sensor occupying the remaining 57.0%.

We show an energy-efficient redesign of a traditional camera sensor network to a more storage-centric network where the use of pull-based techniques give us the improved efficiency over the traditional push-based approach. We also demonstrate that using Capsule in sensor applications is feasible and extremely energy-efficient.

8. RELATED WORK

To our knowledge, our previous work [Mathur et al. 2006a] is the only published work to date comparing different alternatives for energy-efficient local flash storage in sensor networks, or the storage, computation and communication trade-offs that emerge as energy costs of storage decrease sharply.

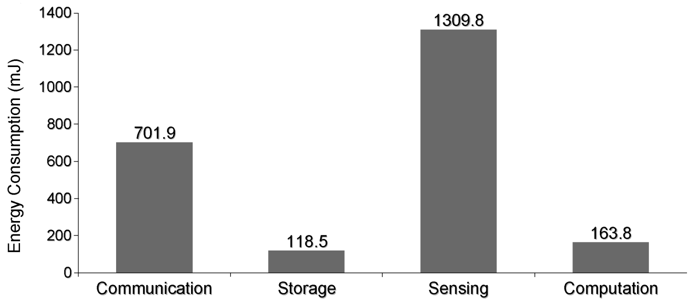

Fig. 14. Component level breakdown of the MicaZ sensor equipped with the Cyclops and local NAND flash storage. Each image is stored locally and a sub-sampled image is sent to the base-station as the event summary. Based on the summary, if the full-resolution image is requested then the sensor performs a wavelet transform on the image and uses run-length encoding to reduce the image size. The image is then chopped into packet-sized fragments and stored on flash and then transmitted using the CC2420 radio.

A few studies have quantified the energy consumption of individual flash memory devices chosen as part of sensor platform designs. The RISE project [Mitra et al. 2005] at UC Riverside has developed a sensor platform with an interface to external SD/MMC flash storage. They present measurements [Zeinalipour-Yazti et al. 2005] of energy consumption for a single SD card, which are comparable to those for one of the less efficient MMC cards we tested. Several other studies have quantified the energy consumption of currently available flash storage on motes. Accurate energy figures for the Mica flash are presented as part of the energy budget planning for the Great Duck Island deployment [Mainwaring et al. 2002], and power consumption data is available for the flash storage on the Telos mote [Polastre et al. 2005].

Other studies have compared energy use and performance of flash storage technologies in the context of handheld battery-powered devices [Park et al. 2004; Lee and Chang 2003]. Since these devices are typically powered by rechargeable batteries, power consumption is often of secondary importance compared to performance; the results of these studies are therefore not directly applicable to sensor platforms.

There have been four other efforts at building a sensor storage system that we are aware of: Matchbox [Gay 2003], ELF [Dai et al. 2004], MicroHash [Zeinalipour-Yazti et al. 2005], and TFFS [Gal and Toledo 2005].[4] We compare Capsule to these systems, discussing issues of energy efficiency, portability, and functionality:

Energy Efficiency. Of the systems that we compare, only MicroHash and Capsule make claims about energy efficiency. MicroHash provides an implementation of a Stream and Index object for SD-cards with greater emphasis on the indexing and lookup techniques than in our paper. A fundamental difference

[4]Other filesystems like YAFFS2 [Manning 2002] and JFFS2 [Woodhouse 2001] are not considered, as they are targeted at portable devices such as laptops and PDAs, and do not have sensor-specific implementations.

between the two systems is that MicroHash uses a page buffer for reads as well as writes, and does not provide the ability to tune the chunk size to the access pattern. This is unlike our system, which can adapt the choice of the chunk sizes to the insert and lookup patterns, thereby better optimizing energy efficiency (Section 6.2.2).

Portability. Embedded platform design is an area of considerable churn, as evident from the plethora of sensor platforms that are being developed and used by research groups. Storage subsystems for these platforms differ in the type of flash (NAND or NOR), page size (256b to 4096b), erase block size (256b to 64KB), bus speeds (SPI or parallel), and energy consumption. It is therefore essential to design a general purpose storage system that can be easily ported to a new platform with a new storage subsystem, while being sufficiently flexible to enable developers to take advantage of new architectures. We believe Capsule achieves these dual goals—it currently works on the Mica2, Mica2dot (both NOR) and our custom NAND board. In contrast, MicroHash is only available for the RISE platform, while Matchbox and ELF work only on the Mica2 NOR flash.

Functionality. MicroHash is the closest existing system to Capsule, with functionality similar to that of the Capsule Index object. Like the Capsule Index, MicroHash uses a fixed-range index; however its index has only a single level; as the number of entries in an index bucket grows, additional directory pages are linked together. Capsule is thus able to look up and retrieve an item with much less overhead, and requires much less RAM buffering. In contrast to Capsule, MicroHash is able to index and retrieve multiple elements with the same key, and uses its index storage somewhat more efficiently when sparsely populated. Finally, the Capsule Stream/Index object offers more powerful navigation methods than MicroHash.

More generally, we note that in comparison to the research effort that has gone into the design of the radio stack on sensors, there have been relatively few efforts at building the sensor storage system. As storage becomes a more important part of sensor network design, increased attention is needed to address questions of storage capacity, failure handling, long-term use, and energy consumption that are not addressed by existing efforts. Capsule attempts to fill this gap by building up a functionally complete storage system for sensors.

Our work is related to a number of object and file systems outside of the field of sensor networks, as well, such as the log-structured file system LFS [Rosenblum and Ousterhout 1992], and Vagabond [Nrvag 2000], a temporal log-structured object database. These systems, however, are disk-based systems designed for read-write optimization, security, and network sharing. Capsule, however, is designed specifically for sensor platforms using NAND or NOR flash memory based storage and optimized for energy efficiency. This results in substantially different technical strategies; for instance, the compaction techniques used in Capsule significantly different from storage reclamation techniques found to be effective for disks, such as hole-plugging [Wilkes et al. 1996] and heuristic cleaning [Blackwell et al. 1995].

9. CONCLUSIONS

In this article we argue that a simple file system abstraction is inadequate for realizing the full benefits of flash storage in data-centric applications. Instead, we advocate a rich object storage abstraction to support flexible use of the storage system for a variety of application needs, and one which is specifically optimized for memory and energy-constrained sensor platforms. We proposed *Capsule*, an energy-optimized log-structured object storage system for flash memories that enables sensor applications to exploit storage resources in a multitude of ways. Capsule employs a hardware abstraction layer to hide the complexities of flash memory from the application, and supports highly energy-optimized implementations of commonly used storage objects such as streams, files, arrays, queues and lists. Further, Capsule supports checkpointing and rollback to tolerate software faults in sensor applications running in unreliable environments. Our Capsule implementation is portable, and currently supports the Mica2 and Mica2Dot NOR flash as well as our custom-built NAND flash memory board. We also showcase the use of Capsule in our deployment of a storage-centric camera sensor network. Here our experiments have demonstrated that Capsule provides greater functionality, more tunability, and greater energy-efficiency than existing sensor storage solutions, while operating within the resource constraints of the Mica2.

Future Work. We plan to examine the platform-specific design of Capsule in light of more resource-rich platforms such as the iMote2. For example, a memory-rich platform would allow Capsule to use a per-object log-segment allocation strategy that would place each object's data chunks contiguously, permitting FAL to do read-buffering. In additional, even on memory-limited platforms it may be possible to achieve improvements in performance with the application of small amounts of additional memory. Finally, we are also working on fabricating an SPI based NAND flash daughter board for the Telos.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for Mathur et al. [2006a, 2006b] and for their valuable feedback and comments.

REFERENCES

AGILENT. Cyclops camera. `http://www.cyclopscamera.org`.

BLACKWELL, T., HARRIS, J., AND SELTZER, M. 1995. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 277–288.

CROSSBOW TECHNOLOGY. 2004. *Datasheet: MICA2*. Crossbow Technology. part number 6020-0042-06 Rev A.

DAI, H., NEUFELD, M., AND HAN, R. 2004. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04)*. ACM Press, New York, NY, USA, 176–187.

GAL, E. AND TOLEDO, S. 2005. A transactional flash file system for microcontrollers. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 89–104.

GAY, D. 2003. Design of matchbox, the simple filing system for motes. in TinyOs 1.x distribution, `www.tinyos.net`. Version 1.0.

HELLERSTEIN, J., HONG, W., MADDEN, S., AND STANEK, K. 2003. Beyond average: Towards sophisticated sensing with queries. In *Proceedings of the International Conference on Information Processing on Sensor Networks (IPSN'03)*.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. *ACM SIGPLAN Notices 35*, 11, 93–104.

HUI, J. W. AND CULLER, D. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*.

LEE, H. G. AND CHANG, N. 2003. Energy-aware memory allocation in heterogeneous non-volatile memory systems. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED'03)*. 420–423.

LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. 2005. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence*, Springer-Verlag.

LI, M., GANESAN, D., AND SHENOY, P. 2006. Presto: Feedback-driven data management in sensor networks. In *Proceedings of the 3nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'06)*.

MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. 2005. Tinydb: An acqusitional query processing system for sensor networks. *ACM Trans. Datab. Syst*.

MAINWARING, A., POLASTRE, J., SZEWCZYK, R., CULLER, D., AND ANDERSON., J. 2002. Wireless sensor networks for habitat monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*. Atlanta, GA.

MANNING, C. 2002. YAFFS: the NAND-specific flash file system. www.aleph1.co.uk/yaffs.

MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. 2006b. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. 2006a. Ultra-low power data storage for sensor networks. In *Proceedings of the 5th International Conference on IPSN/SPOTS*.

MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. 1997. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the 16th Symposium on Operating Systems Principles*. 238–251.

MITRA, A., BANERJEE, A., NAJJAR, W., ZEINALIPOUR-YAZTI, D., GUNOPULOS, D., AND KALOGERAKI, V. 2005. High performance, low power sensor platforms featuring gigabyte scale storage. In *Proceedings of the 3rd International Workshop on Measurement, Modeling, and Performance Analysis of Wireless Sensor Networks (SenMetrics'05)*.

NIGHTINGALE, E. B. AND FLINN, J. 2004. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*.

NRVAG, K. 2000. Vagabond: The design and analysis of a temporal object database management system. Tech. Rep., PhD thesis, Norwegian University of Science and Technology.

PARK, C., KANG, J.-U., PARK, S.-Y., AND KIM, J.-S. 2004. Energy-aware demand paging on NAND flash-based embedded storages. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'04)*. 338–343.

POLASTRE, J., SZEWCZYK, R., AND CULLER, D. 2005. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Conference on IPSN/SPOTS*.

POTTIE, G. J. AND KAISER, W. J. 2000. Wireless integrated network sensors. *Comm. ACM 43*, 5, 51–58.

RAMANATHAN, N., CHANG, K., KAPUR, R., GIROD, L., KOHLER, E., AND ESTRIN., D. November 2-4, 2005. Sympathy for the sensor network debugger. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

RATNASAMY, S., ESTRIN, D., GOVINDAN, R., KARP, B., S. SHENKER, L. Y., AND YU, F. 2001. Data-centric storage in sensornets. In *Proceedings of the 1st ACM Workshop on Hot Topics in Networks*.

RATNASAMY, S., KARP, B., YIN, L., YU, F., ESTRIN, D., GOVINDAN, R., AND SHENKER, S. 2002. GHT—a geographic hash-table for data-centric storage. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and their Applications*.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

TOSHIBA AMERICA ELECTRONIC COMPONENTS, INC.. 2003. Datasheet: TC58DVG02A1FT00. Toshiba America Electronic Components, Inc. (TAEC), www.toshiba.com/taec.

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. 1996. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1, 108–136.

Woodhouse, D. 2001. Journalling Flash File System. In *Proceedings of the Ottowa Linux Symposium*.

Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. 2005. MicroHash: An efficient index structure for flash-based sensor devices. In *Proceedings of the 4th USENIX FAST Conference*.

Received October 2007; revised August 2008; accepted October 2008